Exploitation d'accélérateurs pour le calcul de fonctions de structure de champs scalaires discontinus

Fabien Thiesset (CORIA, Rouen),

Alexandre Poux (CORIA, Rouen),

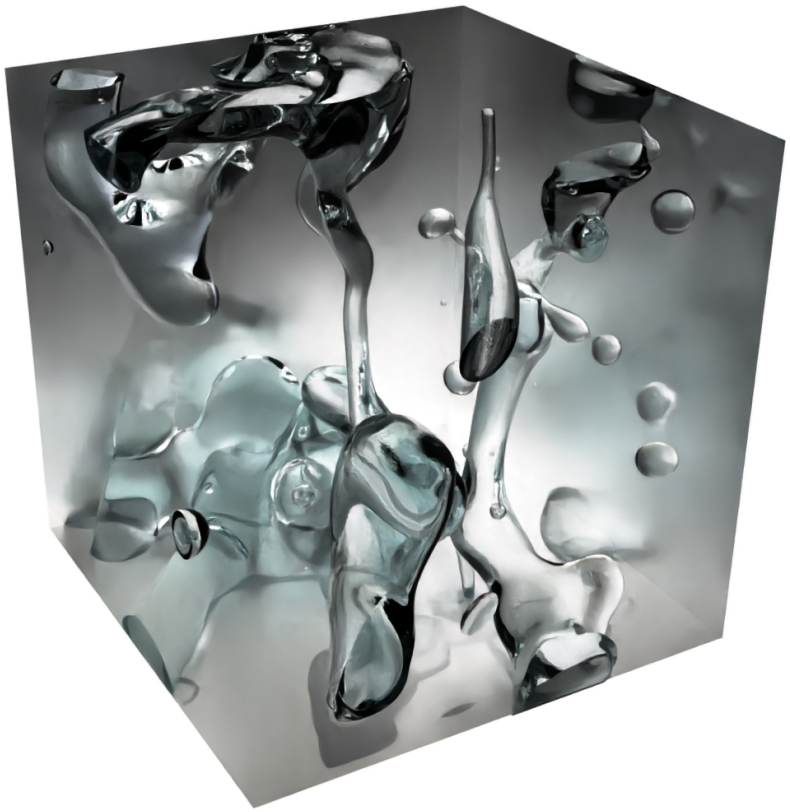
Patrick Bousquet-Mélou (CRIANN, Rouen)

CRIANN

Contexte : les écoulements turbulents diphasiques

- Simulations numériques d'un écoulement de fluides non-miscibles

CRIANN



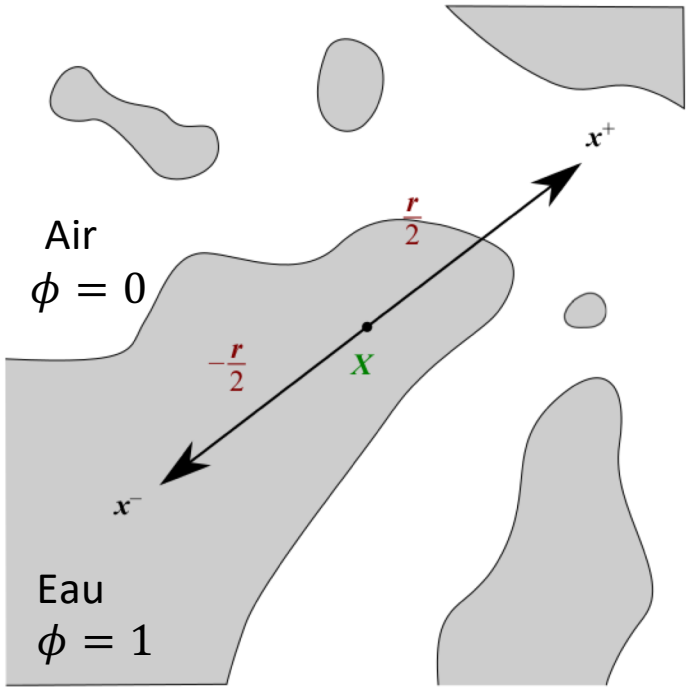
Exemple :
- Eau/Air (Océan-Atmosphère)
- H2 liquide

Question scientifique :

Quelle est la forme des structures fluides ?

-> Indicateur morphologique

Fonctions de structure de champs scalaires discontinus



La morphologie d'un champ discontinu ϕ peut être mesurée grâce à sa fonction de structure

$$\langle(\delta\phi)^2\rangle(\vec{r}) = \left\langle\left(\phi_{\vec{x}+\vec{r}} - \phi_{\vec{x}}\right)^2\right\rangle$$

où les crochets $\langle \quad \rangle$ désignent une moyenne sur $\vec{x}$.
Notons que

$$\langle(\delta\phi)^2\rangle = \left\langle\phi^2_{\vec{x}+\vec{r}}\right\rangle + \left\langle\phi^2_{\vec{x}}\right\rangle - 2\left\langle\phi_{\vec{x}+\vec{r}}\phi_{\vec{x}}\right\rangle$$

où $\left\langle\phi_{\vec{x}+\vec{r}}\phi_{\vec{x}}\right\rangle$ est la fonction de corrélation

Les corrélations peuvent être calculées grâce aux **FFT/IFFT** (N log N) seulement si
- le champ est périodique ... pas toujours le cas
- le champ est continu (sinon erreur de troncature) ... clairement pas le cas pour ϕ

Pas d'autres moyens que de passer par la force brute (N^6)

Objectif

Exploiter des accélérateurs avec un code Fortran lisible et maintenable par un non-expert :

- Programmation par directive
- Pas d'optimisations excessives

Austral (HPE)

- 24 768 cœurs CPU de calcul AMD **EPYC 9654** (**Genoa**, 96-Core par socket)
  - Intel compiler 23.0 + OpenMP
  - Gnu compiler 12.2.1 + OpenMP
  - Cray compiler 17.0.1 + OpenMP
- 88 GPU NVIDIA **Ampere A100**
  - nvhpc 23.11 + OpenACC ou «OpenMP target»
- 8 GPU AMD **MI210**
  - Cray compiler 17.0.1 + rocm 6.1.3 + + OpenACC (Fortran) ou «OpenMP target»

Boréale (NEC)

- 72 Vector Engines NEC SX Aurora Tsubasa 20B
  - NEC Fortran Compiler (4.0.0) for Vector Engine

Code

- Parallélisation par thread OpenMP (+ vectorisation) ou gang openacc

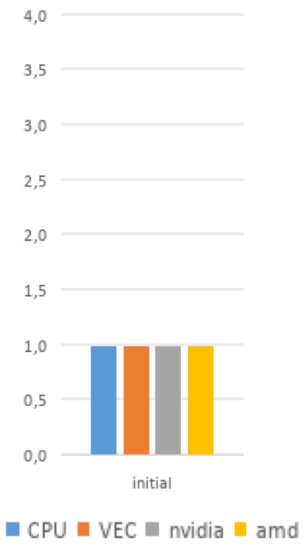
```
!$omp parallel do collapse(3) default(none) &
!$omp shared(phi, r, steps, nr, cnt, n_phi) shared(result3d)
!$acc parallel loop gang collapse(3) copyin(phi, r, steps, nr, cnt, n_phi) copyout(result3d)
do kr = 1, nr
do jr = 1, nr
do ir = 1, nr
   result3d(ir, jr, kr) = da2(phi, r(ir:ir), r(jr:jr), r(kr:kr), steps, n_phi, cnt)
end do
end do
end do
!$acc end parallel loop
!$omp end parallel do
```

- Travail sur variantes d'implémentation de la fonction **da2** sur les différentes architectures

Code

```
pure function da2_kji_initial(phi, ri_, rj_, rk_, steps, n_phi, cnt) result(da2)
  !$acc routine vector
  implicit none

  integer, intent(in) :: n_phi(3) ! no assumed shape
  real(kind=sp), intent(in) :: phi(n_phi(1), n_phi(2), n_phi(3))
  integer, intent(in) :: ri_(1), rj_(1), rk_(1), steps(3)
  integer, intent(in) :: cnt
  real(kind=sp) :: da2
  integer :: ip, jp, kp, im, jm, km, ri, rj, rk

  ! fix for OpenAcc with Cray Fortran
  ri = ri_(1) ; rj = rj_(1) ; rk = rk_(1)

  da2 = 0
  do km = 1, n_phi(3), steps(3)
    kp = modulo(km + rk-1, n_phi(3))+1
    do jm = 1, n_phi(2), steps(2)
      jp = modulo(jm + rj-1, n_phi(2))+1

#if !defined(__NEC__) && !defined(_OPENACC)
        ! Simd reduction sur CPU / vectorisation automatique sur Vector Engine
        !$omp simd reduction(+:da2)
#endif
        do im = 1, n_phi(1), steps(1)
          ip = modulo(im + ri-1, n_phi(1))+1

          da2 = da2 + (phi(ip, jp, kp) - phi(im, jm, km))**2
        end do
      end do
    end do
    da2 = da2/cnt
  end function da2_kji_initial
```

- Version initiale multi-architecture



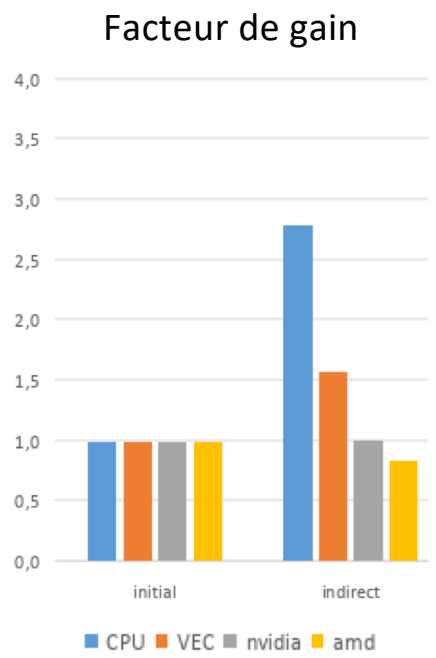
Code

- Idée 1 : précalculer les modulos

```
do kr = 1, nr
do k = 1, n_red(3)
   kkm(k) = steps(3)*(k - 1) + 1
   kp = kkm(k) + r(kr)
   if (kp > n_phi(3)) then
      kp = kp - n_phi(3)
   else if (kp < 1) then
      kp = kp + n_phi(3)
   end if
   kkp(k, kr) = kp
end do
end do
```

-> moins d'opérations dans le noyau de calcul, mais des accès indirects aux indices de phi y restent

```
da2 = 0
!$acc loop vector reduction(+:da2)
do k = 1, n_red(3)
   km = kkm(k)
   kp = kkp(k)
   do j = 1, n_red(2)
      jm = jjm(j)
      jp = jjp(j)
#if !defined(__NEC__) && !defined(_OPENACC)
         !$omp simd reduction(+:da2)
#endif
         do i = 1, n_red(1)
            im = iim(i)
            ip = iip(i)
            da2 = da2 + (phi(ip, jp, kp) - phi(im, jm, km))**2
         end do
      end do
   end do
   da2 = da2/cnt
```



- Non avantageux sur GPU
- Avantageux sur Vector Engine
- Très avantageux sur CPU

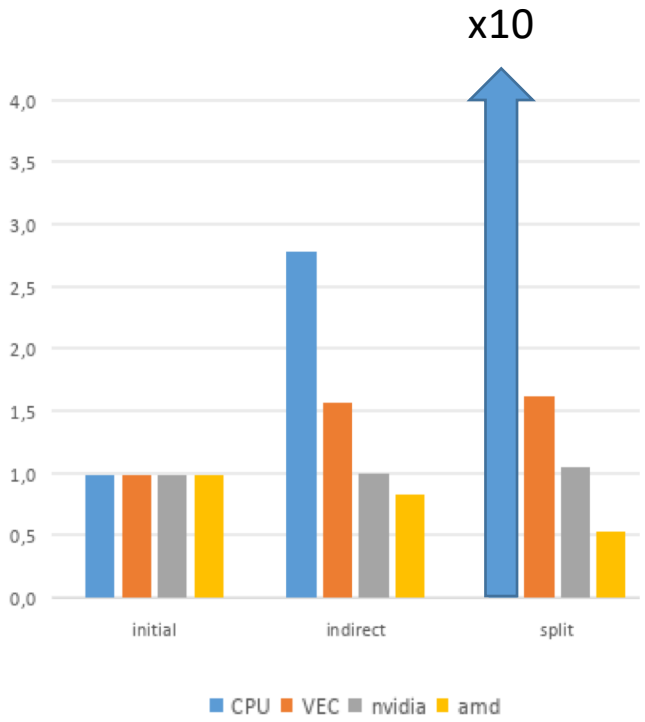
Code

- Idée 2 : accès directs aux indices de phi

Possible en découpant la boucle interne

```
      da2 = 0
      !$acc loop vector reduction(+:da2)
      do km = 1, n_phi(3), steps(3)
        kp = modulo(km + rk-1, n_phi(3))+1
        do jm = 1, n_phi(2), steps(2)
          jp = modulo(jm + rj-1, n_phi(2))+1

#if !defined(__NEC__) && !defined(_OPENACC)
          !$omp simd reduction(+:da2)
#endif
          do im = i1s, i1e, steps(1)
            da2 = da2 + (phi(im + i1p, jp, kp) - phi(im, jm, km))**2
          end do

#if !defined(__NEC__) && !defined(_OPENACC)
          !$omp simd reduction(+:da2)
#endif
          do im = i2s, i2e, steps(1)
            da2 = da2 + (phi(im + i2p, jp, kp) - phi(im, jm, km))**2
          end do

#if !defined(__NEC__) && !defined(_OPENACC)
          !$omp simd reduction(+:da2)
#endif
          do im = i3s, i3e, steps(1)
            da2 = da2 + (phi(im + i3p, jp, kp) - phi(im, jm, km))**2
          end do
        end do
      end do
      da2 = da2/cnt
    end function da2_kji_split
```

Facteur de gain



- Efficace sur Vector Engine
- Très efficace sur CPU

Code GPU

- Idée 3 : coalescence de mémoire / boucle vectorisée sur GPU

```
pure function da2_kji_indirect_implciit_reduction (phi, iip, &
   jjp, kkp, iim, jjm, kkm, n_phi, n_red, cnt) result(da2)

      !$acc routine vector

      ! [declarations]

      da2 = 0
      do k = 1, n_red(3)
         km = kkm(k)
         kp = kkp(k)
         do j = 1, n_red(2)
            jm = jjm(j)
            jp = jjp(j)
            do i = 1, n_red(1)
               im = iim(i)
               ip = iip(i)
               da2 = da2 + (phi(ip, jp, kp) - phi(im, jm, km))**2
            end do
         end do
      end do
      da2 = da2/cnt
   end fucntion da2_kji
```

nvfortran -fast -**Minfo** -**acc** -gpu=cc80 -c \
increments_fortran.F90

```
da2_kji_indirect_implicit_reduction:
   201, Generating NVIDIA GPU code
      216, !$acc loop seq
           Generating implicit reduction(+:da2)
      219, !$acc loop seq
           Generating implicit reduction(+:da2)
      222, !$acc loop vector ! threadidx%x
           Generating implicit reduction(+:da2)
           Vector barrier inserted for vector loop reduction
   216, Loop is parallelizable
   219, Loop is parallelizable
   222, Loop is parallelizable
        Generated vector simd code for the loop
containing reductions
   225, FMA (fused multiply-add) instruction(s)
generated
```

Le compilateur vectorise la boucle interne.
Ici c'est celle ayant un accès contigu à la mémoire ; la coalescence de mémoire optimise alors la bande passante mémoire

Code GPU

- Idée 3 : coalescence de mémoire / boucle vectorisée sur GPU (suite)

```
pure function da2_ikj_indirect (phi, iip, &
   jjp, kkp, iim, jjm, kkm, n_phi, n_red, cnt) result(da2)

      !$acc routine vector
      ! [declarations]

       da2 = 0
      !$acc loop vector reduction(+:da2)
       do i = 1, n_red(1)
          im = iim(i)
          ip = iip(i)
           do k = 1, n_red(3)
             km = kkm(k)
             kp = kkp(k)
             do j = 1, n_red(2)
                jm = jjm(j)
                jp = jjp(j)
                 da2 = da2 + (phi(ip, jp, kp) - phi(im, jm, km))**2
            end do
          end do
       end do
       da2 = da2/cnt
   end fucntion da2_kji
```

```
nvfortran -fast -Minfo -acc -gpu=cc80 -c \
      increments_fortran.F90
```

```
da2_ikj_indirect:
   233, Generating NVIDIA GPU code
      249, !$acc loop vector ! threadidx%x
         Generating reduction(+:da2)
      252, !$acc loop seq
      255, !$acc loop seq
      260, Vector barrier inserted for vector loop
reduction
   249, Loop is parallelizable
   252, Loop is parallelizable
   255, Loop is parallelizable
      Generated vector simd code for the loop containing
reductions
   258, FMA (fused multiply-add) instruction(s) generated
```

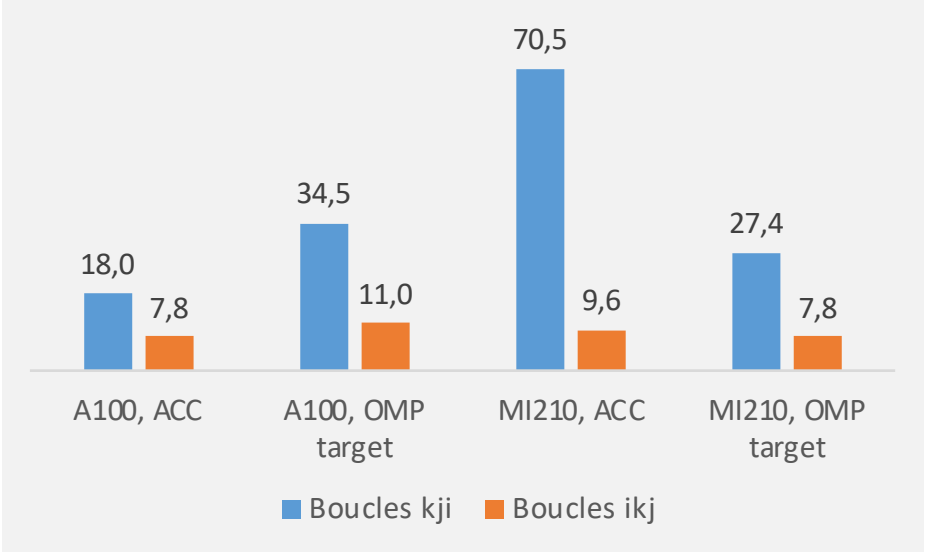
Sur GPU ne vaut-il pas mieux vectoriser la boucle externe de de code ?
Cette version teste cette option (boucle i externe)

Code GPU

- Idée 3 : coalescence de mémoire / boucle vectorisée sur GPU (fin)

Sur GPU ne vaut-il pas mieux **vectoriser la boucle externe** de la fonction de calcul de cette application ?
Pour tester cela, la boucle sur le premier indice (i) est remontée à l'extérieur (pour la coalescence de mémoire sur GPU, c'est cette boucle qu'il faut vectoriser pour optimiser La bande passante mémoire)

Temps elapsed (sec), OpenACC/OpenMP target
Maillage 128^3



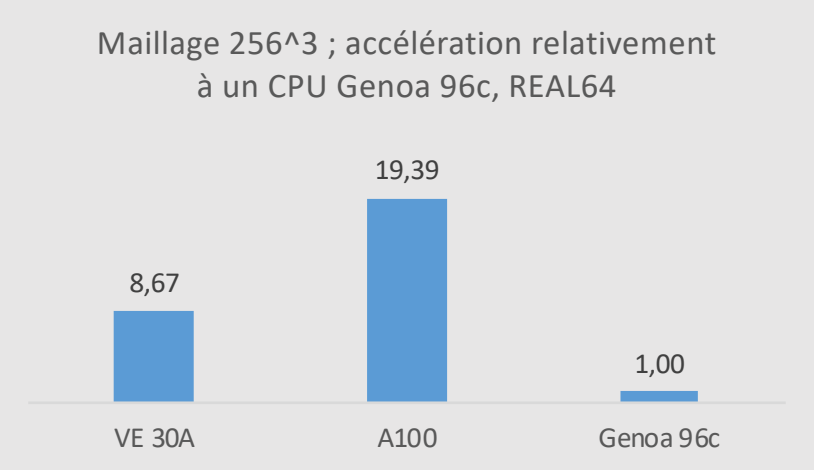
Co-processeurs vectoriels

- Dans le cadre du partenariat NEC/CRIANN (MesoNET), NEC a exécuté l'application sur la dernière génération de Vector Engine (VE A30)

- NEC VE A30 : 16 cœurs, vecteurs de 256x(64 bits)
  - 4,9 TFlops DP, 2,45 TB/s

- Versus NVIDIA A100-SXM4-80GB
  - 9,7 TFlops DP, 2 TB/s

  - L'intensité arithmétique de l'application favorise le GPU
  - Gain VE / CPU non négligeable
  - Programmation standard d'un VE : threads OpenMP et vectorisation automatique de boucle interne (vs OpenACC ou « OpenMP target » sur A100)



Conclusion

- L’optimisation pour architectures multiples conduirait à un code multi-versions
- L’équipe peut préférer retenir la variante offrant le meilleur compromis, pour sa production au moyen de cette application de post-traitement

Remerciements
- Rene Puttin de NEC Deutschland GmbH
- Ce travail a bénéficié d'un accès au centre de ressources MesoNET et au projet MesoNET sous l'allocation M24074.